RCU API

Author: Madhuparna Bhowmik (madhuparnabhowmik04@gmail.com)

In this article, we will see how we can use RCU API to delete and update nodes in a list with the help of pseudo codes.

Consider a linked list where each node is defined like this:

```
struct Node{
  int data;
  struct Node *next;
}
```

Let's say the list has 4 nodes, A,B,C,D

A --> B --> C --> D --> NULL

1. Delete

To delete node B from this list, we can do the following:

```
1. A->next = B->next;
2. free(B)
```

Now, if there are multiple readers reading this list and a writer wants to delete node B from the list, then the simplest way to achieve consistency is to allow either the writer to do the delete operation or allow the readers to read concurrently. However, the writer would have to wait for all the pre-existing readers to finish to continue with the delete operation.

But as we will see, using RCU, the writer can start doing the delete operation without waiting for the readers to finish.

Suppose, we do the following:

```
1. A->next = B->next;
2. wait until all readers that were already processing the list finish.
3. free(B)
```

In the above pseudo code, before free(B) is called, the list will look like this:

```
A --> C --> D --> NULL
  ->
 B-
```

That is, B is still pointing to C.

So, if a reader is processing Node B and this update is made by the writer, even then the reader will see a consistent list. And any new readers will only see the new list and cannot get a reference to node B. So, we have two versions of the list now.

Therefore when all the readers that were reading the list before the writer started manipulating the pointers, finish, we are assured that all the other readers are reading the new version of the list, and therefore, the

writer can safely free node B.

```
A --> C --> D --> NULL
```

One important point to note here is that in all systems running Linux, loads from and stores to pointers are atomic. Therefore, A->next = B->next does not introduce any inconsistency.

Next, let's see how we can use RCU API to implement the above written pseudo code:

Readers's Code:

```
1. rcu_read_lock();
2. Do some read operation
3. rcu_read_unlock();
```

Writer's Code:

```
1. spin_lock(&mylock);
2. A->next = B->next
3. synchronize_rcu();
4. free(B);
5. spin_unlock(&mylock);
```

In the pseudo code for readers, any number of readers can read the list concurrently,therefore adding the code within the rcu_read_lock() and rcu_read_unlock() will keep track of readers.

In the pseudo code for writers, only one writer should be allowed to enter the critical section so we need a spinlock to enforce this condition.

Using synchronize_rcu() the writer waits for all the readers that had started with the read operation before the writer, to finish. After this is over, the writer can safely free B.

2. Update

In this section we will see how we can modify the data in a node by keeping a copy of it.

Suppose, we want to modify B->data from 5 to 7. Then, we can do the following:

1. Create a node new_node of type struct Node.
2. Copy B to new_node.
3. Do the update operation on the new_node, i.e., do

   ```
   new_node->data = 7
   ```

   in this case.
4. Make new_node visible to the readers by changing A's next to new_node.

   ```
   A->next = new_node
   ```

5. Wait for pre-existing readers to finish.
6. Free Node B.

Pseudo Code:

1. new_node = kmalloc(sizeof(*B), GFP_KERNEL);
2. *new_node = *B
3. new_node->data = 7
4. A->next = new_node
5. synchronize_rcu();
6. kfree(B)

Thank you!

References

1. What is RCU, Fundamentally? Part 1 by Paul McKenny

2. The RCU API, 2019 edition by Paul Mckenny

3. What is RCU in Linux- quora answer by Siddharth Teotia